

MAULANA ABUL KALAM AZAD
UNIVERSITY OF TECHNOLOGY,
WEST BENGAL



Maulana Abul Kalam Azad University Of Technology

Wireless and Sensor Networks Laboratory [PGIT(IOT)292]

Submitted by

Subhadip Manna

[Reg. No: 233000410850]

[Roll No : 30060823005]

Subject: Wireless and Sensor networks

Course : **M.Tech**(Internet of Things)

Semester : 2ndSemester

Session: 2024-2025

Lab Report Submitted to

Course Faculty

Kamalika Bhattacharya , Department Of InformationTechnology

Table of Contents

Exp.No.	Name Of the Experiment	Page No.	Date Of Experiment	Date Of submission	Signature
I	Setup your Network simulator environment using python and V.S Code to Run and Test Wireless Sensor Network Experiments.	1 - 3			
1	Write a program in Wireless Sensor Network to create a network model and show how to data transfer and communicate between nodes.	4 - 10			
2	Write a program Implement CSMA/CD MAC protocol in Wireless Sensor Network Using Python.	11 - 17			
3	Write a program Implement Schedule Based MAC protocol like TDMA , FDMA In Wireless Communication Using Python.	18 - 23			
4	Simulating and Testing Routing Protocols in Wireless Sensor Networks (WSN).	24 - 34			
5	Simulating and Testing Leach Routing Protocols in Wireless Sensor Networks (WSN).	35 - 39			

I: Setup your Network simulator environment using python and V.S Code to Run and Test Wireless Sensor Network Experiments.

Answer: Following are steps to Setup –

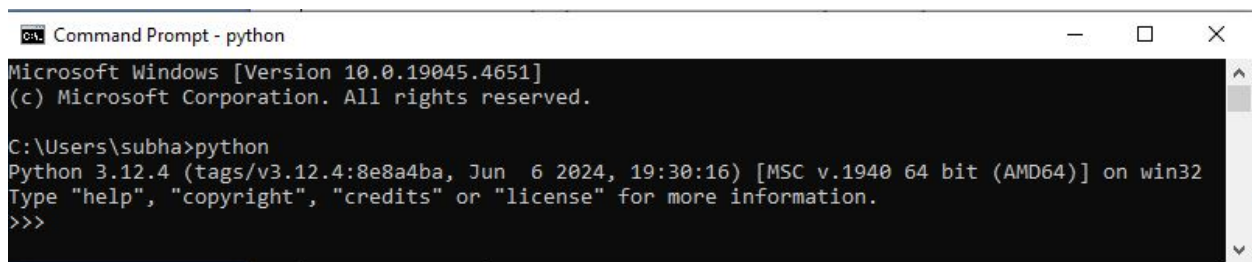
Step 1 : Download and Install Python: Go to the official website link –

<https://www.python.org/downloads/> Download latest version python according to your system (In my Case I am chose python 3.12.4 windows 64 bit operating system).

After download are complete . For the installation go to the official website

<https://docs.python.org/3/using/windows.html> follow the steps to install python.

Open Command Prompt and type ***python***



```
Command Prompt - python
Microsoft Windows [Version 10.0.19045.4651]
(c) Microsoft Corporation. All rights reserved.

C:\Users\subha>python
Python 3.12.4 (tags/v3.12.4:8e8a4ba, Jun 6 2024, 19:30:16) [MSC v.1940 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

If you got this output then python installation complete .

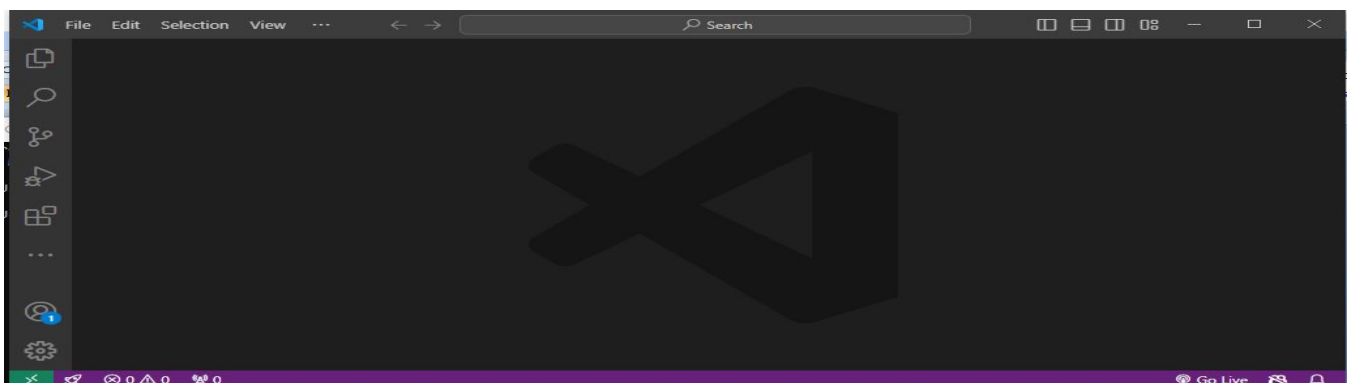
Step 2 : Download and Install VS.Code IDE: Go to the official website link –

<https://code.visualstudio.com/download> Download latest version VS.Code according to your system (In my Case I am chose windows 64 bit operating system).

After download are complete . For the installation go to the official website

<https://www.geeksforgeeks.org/how-to-install-visual-studio-code-on-windows/> follow the steps to

VS.Code . Open Command Prompt and type ***code***



Step3 : Install Pre-Requisite Python Libraries.

NetworkX:

- **Purpose:** Network analysis and graph-based operations.
- **Use:** Creating and manipulating the network topology of WSNs.
- **Installation:** `pip install networkx`

Matplotlib:

- **Purpose:** Data visualization.
- **Use:** Visualizing network topologies and simulation results.
- **Installation:** `pip install matplotlib`

Numpy:

- **Purpose:** Numerical computations.
- **Use:** Handling coordinates and other numerical data in simulations.
- **Installation:** `pip install numpy`

Click:

- **Purpose:** Command-line interface creation.
- **Use:** Building a CLI for running simulations with different configurations.
- **Installation:** `pip install click`

ConfigParser:

- **Purpose:** Configuration file parsing.
- **Use:** Reading simulation parameters from configuration files.
- **Installation:** Part of Python's standard library.

Unittest:

- **Purpose:** Unit testing.
- **Use:** Writing and running tests to ensure the correctness of the simulation code.
- **Installation:** Part of Python's standard library.

Geopy:

- **Purpose:** Geocoding and handling geographical data.
- **Use:** Simulating WSNs with real-world geographical positions.
- **Installation:** `pip install geopy`

Scipy:

- **Purpose:** Advanced scientific computing.
- **Use:** Providing additional tools for mathematical operations and data processing.
- **Installation:** `pip install scipy`

SciPy:

- **Purpose:** Advanced scientific computations.
- **Use:** Additional tools for mathematical operations and data processing.
- **Installation:** `pip install scipy`

SimPy:

- **Purpose:** Process-based discrete-event simulation.
- **Use:** Simulating more detailed and complex events in WSNs.
- **Installation:** `pip install simpy`

Plotly:

- **Purpose:** Interactive visualizations.
- **Use:** Creating interactive plots for WSN data.
- **Installation:** `pip install plotly`

Dash:

- **Purpose:** A framework for building analytical web applications.
- **Installation:** `pip install dash`
- **Usage:** Creates interactive dashboards to visualize real-time data from WSNs.
- **Dash Core Components (`dash-core-components`):**
 - **Purpose:** Provides a set of higher-level components like graphs, sliders, and dropdowns for Dash apps.
 - **Installation:** `pip install dash-core-components`
 - **Usage:** Used to add interactive elements to Dash applications.

Open VS.Code and go to terminal install this Libraries. Now your setup are completed.

Experiment 1: Write a program in Wireless Sensor Network to create a network model and show how to data transfer and communicate between nodes.

Answer:

Explanation:

Wireless Sensor Network Visualization and Communication Simulation

This program simulates a wireless sensor network using NetworkX and Dash. It creates a network of nodes with random connections and visualizes the network using Plotly. The program simulates communication steps between nodes, printing detailed information about the sender, receiver, and the path taken for each communication step.

The stages of communication are:

1. Request: The sender node requests communication with the receiver node.
2. Acknowledgment: The receiver node acknowledges the communication request.
3. Data Transfer: Data is transferred from the sender node to the receiver node.

Each stage is visually represented in the network graph, and the program prints the details of each stage to the console.

Code :

```
import networkx as nx
import plotly.graph_objects as go
import dash
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output
import random
import time
from datetime import datetime, timedelta

# Create a graph for the wireless sensor network
G = nx.Graph()

# Number of nodes
num_nodes = 8

# Add nodes to the graph
nodes = range(num_nodes)
G.add_nodes_from(nodes)
```

```

# Randomly add edges between nodes to simulate wireless connections
edges = [(i, j) for i in nodes for j in nodes if i < j and random.random() > 0.5]
G.add_edges_from(edges)

# Assign positions to nodes for visualization
pos = nx.spring_layout(G)

# Initialize Dash app
app = dash.Dash(__name__)

# State to track the transmission
transmission_state = {
    'step': 0,
    'paths': [],
    'start_time': None,
    'stage': 'idle' # Track the current stage of communication
}

# Function to generate Plotly figure
def create_figure(paths, step, current_time):
    edge_trace = []
    for edge in G.edges():
        x0, y0 = pos[edge[0]]
        x1, y1 = pos[edge[1]]
        edge_trace.append(go.Scatter(
            x=[x0, x1, None], y=[y0, y1, None],
            line=dict(width=0.5, color='#888'),
            hoverinfo='none',
            mode='lines'))

    node_trace = go.Scatter(
        x=[], y=[],
        text=[],
        mode='markers+text',
        textposition='top center',
        hoverinfo='text',
        marker=dict(
            showscale=True,
            colorscale='YlGnBu',
            size=10,
            colorbar=dict(
                thickness=15,
                title='Node Connections',
                xanchor='left',
                titleside='right'
            )
        )
    )

```

```

for node in G.nodes():
    x, y = pos[node]
    node_trace['x'] += tuple([x])
    node_trace['y'] += tuple([y])
    node_trace['text'] += tuple([str(node)])

# Create a figure
fig = go.Figure(data=edge_trace + [node_trace],
    layout=go.Layout(
        title=f'Wireless Sensor Network - Step {step}',
        showlegend=False,
        hovermode='closest',
        margin=dict(b=20, l=5, r=5, t=40),
        xaxis=dict(showgrid=False, zeroline=False),
        yaxis=dict(showgrid=False, zeroline=False)))

# Add arrows for communication paths
for path in paths:
    if isinstance(path, list): # Ensure path is a list
        elapsed_time = (current_time - transmission_state['start_time']).total_seconds()
        if elapsed_time < 50: # Show data transmission for 50 seconds
            for i in range(len(path) - 1):
                x0, y0 = pos[path[i]]
                x1, y1 = pos[path[i+1]]
                fig.add_trace(go.Scatter(
                    x=[x0, x1],
                    y=[y0, y1],
                    mode='markers+lines+text',
                    marker=dict(size=10, color='red'),
                    line=dict(width=2, color='red'),
                    text=[f'{path[i]} -> {path[i+1]}'],
                    textposition='top center'
                ))

    return fig

# App layout
app.layout = html.Div([
    dcc.Graph(id='network-graph'),
    dcc.Interval(
        id='interval-component',
        interval=1*1000, # Update every second
        n_intervals=0
    )
])

@app.callback(

```



```

    Output('network-graph', 'figure'),
    [Input('interval-component', 'n_intervals')]
)
def update_graph(n):
    global transmission_state
    step = transmission_state['step']
    current_time = datetime.now()

    # Only simulate communication steps up to 3 times
    if step < 6:
        if transmission_state['start_time'] is None or (current_time -
transmission_state['start_time']).total_seconds() >= 50:
            source, target = random.sample(nodes, 2)
            path = simulate_communication(G, source, target)
            if path:
                transmission_state['paths'] = [path] # Ensure paths is a list of paths
                transmission_state['start_time'] = current_time
                transmission_state['step'] += 1
                transmission_state['stage'] = 'request'

    # Print messages based on current stage
    if transmission_state['stage'] == 'request':
        path = transmission_state['paths'][0]
        path_str = ' -> '.join(map(str, path))
        print(f"Sender Node {path[0]} is requesting communication with Receiver Node {path[-1]} via path:
{path_str}")
        transmission_state['stage'] = 'acknowledgment'
    elif transmission_state['stage'] == 'acknowledgment':
        elapsed_time = (current_time - transmission_state['start_time']).total_seconds()
        if elapsed_time >= 10: # Wait for 10 seconds to simulate acknowledgment
            path = transmission_state['paths'][0]
            path_str = ' -> '.join(map(str, path))
            print(f"Receiver Node {path[-1]} has granted acknowledgment to Sender Node {path[0]} via path:
{path_str}")
            transmission_state['stage'] = 'data_transfer'
    elif transmission_state['stage'] == 'data_transfer':
        elapsed_time = (current_time - transmission_state['start_time']).total_seconds()
        if elapsed_time >= 20: # Wait for 20 seconds to simulate data transfer
            path = transmission_state['paths'][0]
            path_str = ' -> '.join(map(str, path))
            print(f"Data transfer from Sender Node {path[0]} to Receiver Node {path[-1]} via path: {path_str}")
            print(" ")
            transmission_state['stage'] = 'idle'

    return create_figure(transmission_state['paths'], step, current_time)

# Function to simulate communication steps
def simulate_communication(G, source, target):

```

```

if nx.has_path(G, source, target):
    path = nx.shortest_path(G, source, target)
    return path
else:
    print(f"No path found from Node {source} to Node {target}")
    return []

if __name__ == '__main__':
    app.run_server(debug=True)

```

Output : (console printed)

Sender Node 5 is requesting communication with Receiver Node 6 via path: 5 -> 6
Receiver Node 6 has granted acknowledgment to Sender Node 5 via path: 5 -> 6
Data transfer from Sender Node 5 to Receiver Node 6 via path: 5 -> 6 (figure2).

Sender Node 0 is requesting communication with Receiver Node 7 via path: 0 -> 3 -> 7
Receiver Node 7 has granted acknowledgment to Sender Node 0 via path: 0 -> 3 -> 7
Data transfer from Sender Node 0 to Receiver Node 7 via path: 0 -> 3 -> 7 (figure3).

Sender Node 4 is requesting communication with Receiver Node 7 via path: 4 -> 1 -> 7
Receiver Node 7 has granted acknowledgment to Sender Node 4 via path: 4 -> 1 -> 7
Data transfer from Sender Node 4 to Receiver Node 7 via path: 4 -> 1 -> 7 (figure4).

Sender Node 2 is requesting communication with Receiver Node 5 via path: 2 -> 3 -> 5
Receiver Node 5 has granted acknowledgment to Sender Node 2 via path: 2 -> 3 -> 5
Data transfer from Sender Node 2 to Receiver Node 5 via path: 2 -> 3 -> 5 (figure5).

Sender Node 5 is requesting communication with Receiver Node 2 via path: 5 -> 3 -> 2
Receiver Node 2 has granted acknowledgment to Sender Node 5 via path: 5 -> 3 -> 2
Data transfer from Sender Node 5 to Receiver Node 2 via path: 5 -> 3 -> 2 (figure6)

Sender Node 2 is requesting communication with Receiver Node 7 via path: 2 -> 7
Receiver Node 7 has granted acknowledgment to Sender Node 2 via path: 2 -> 7
Data transfer from Sender Node 2 to Receiver Node 7 via path: 2 -> 7 (figure7)

Visualization output : (Red color represent data transmission between nodes)

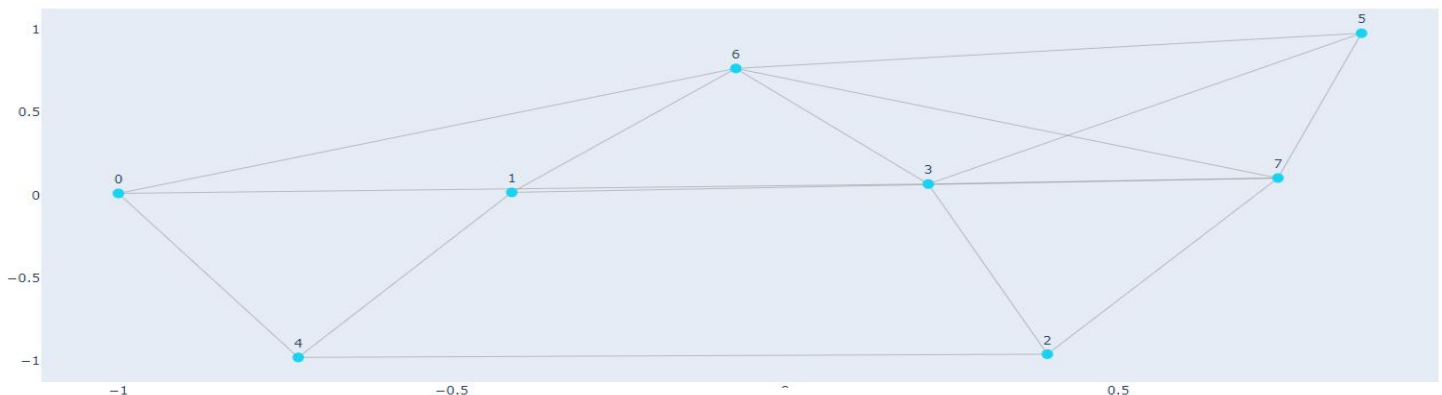


figure:1 (initial network)

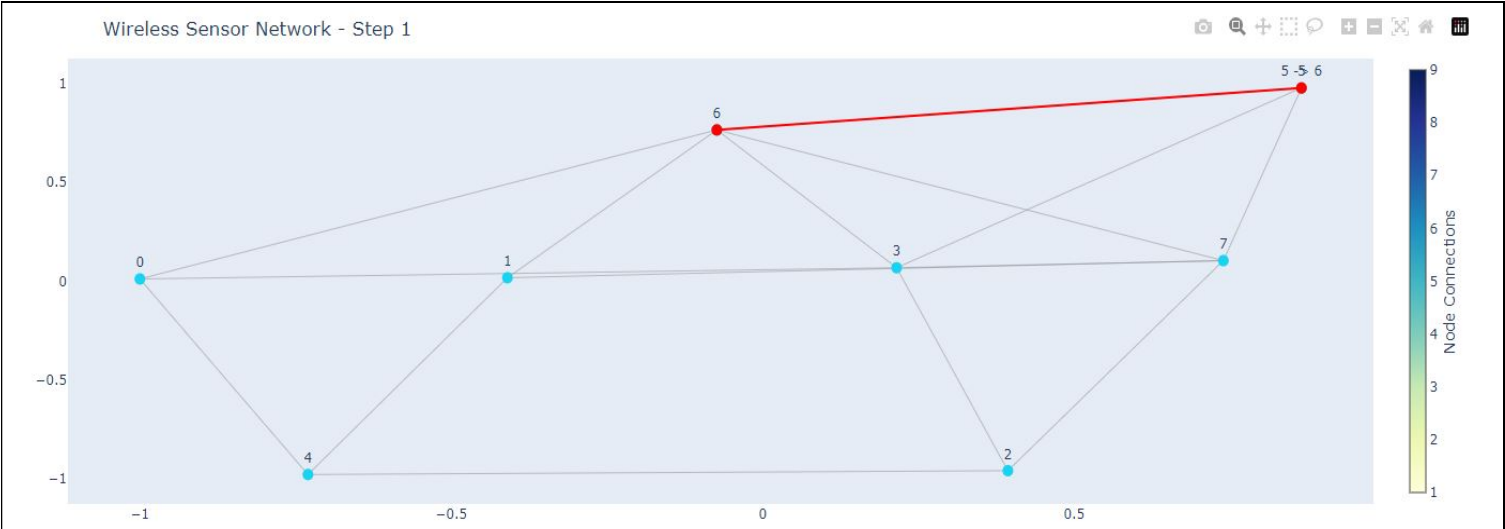


Figure : 2

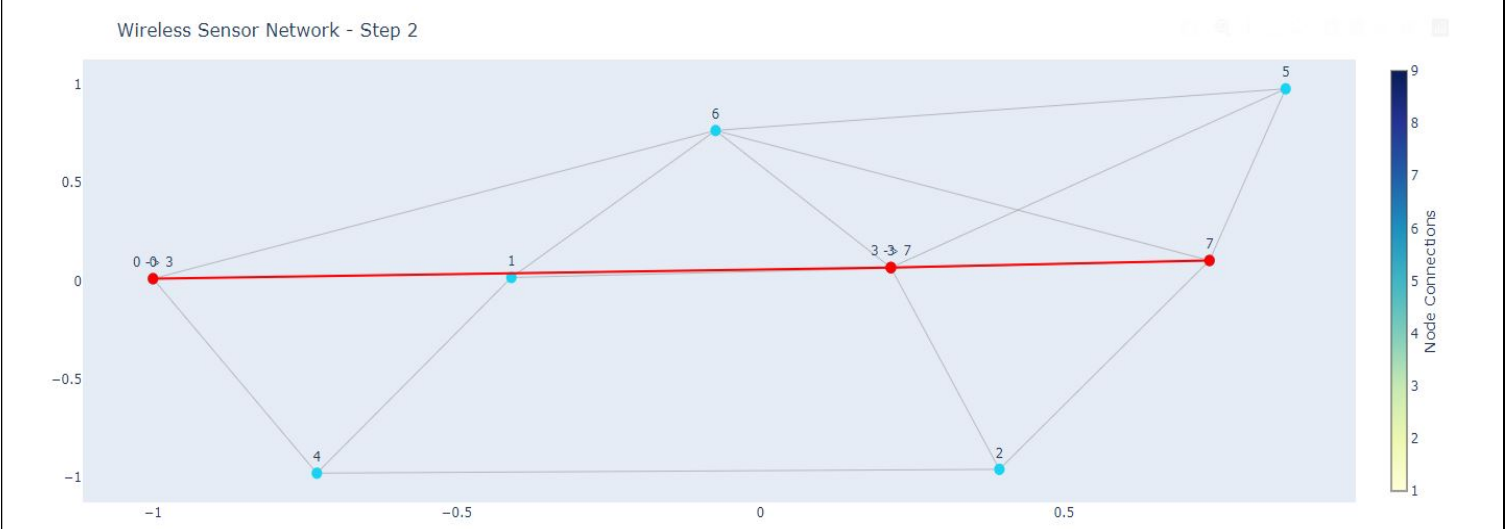


Figure : 3

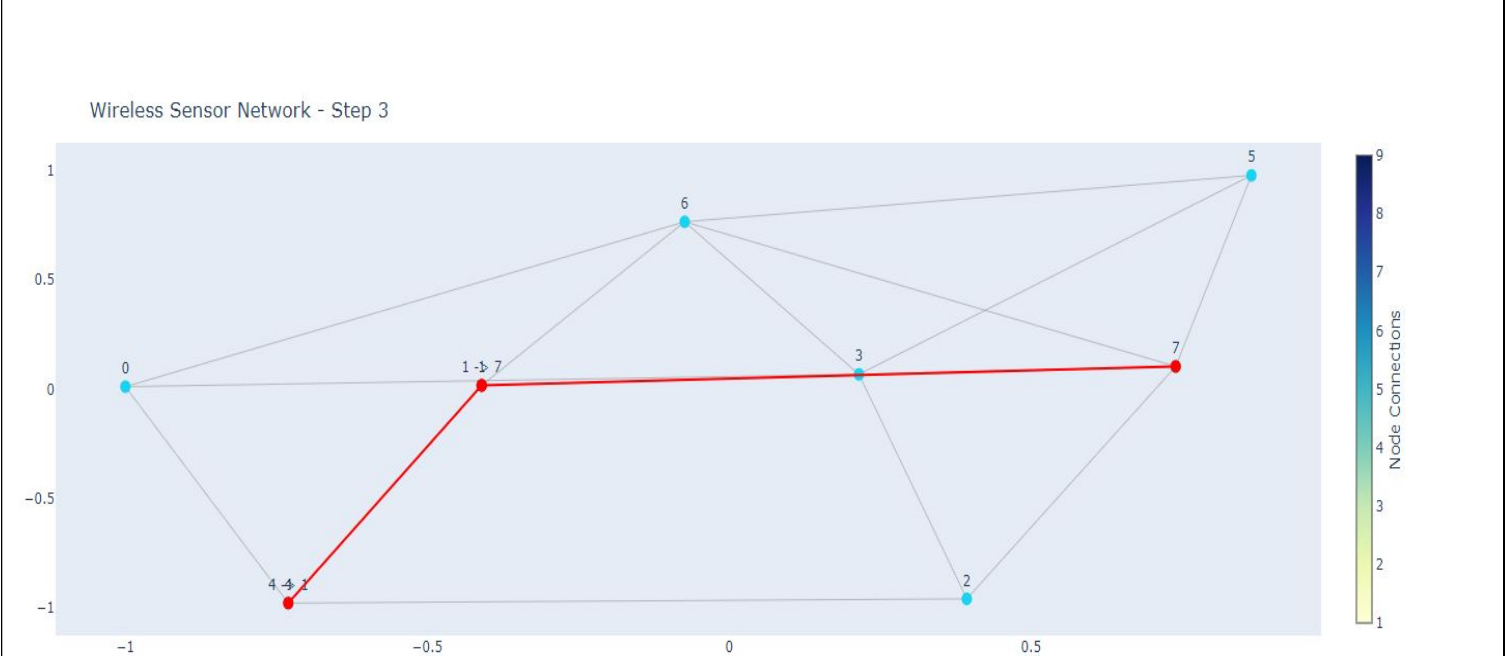


Figure : 4

Wireless Sensor Network - Step 4

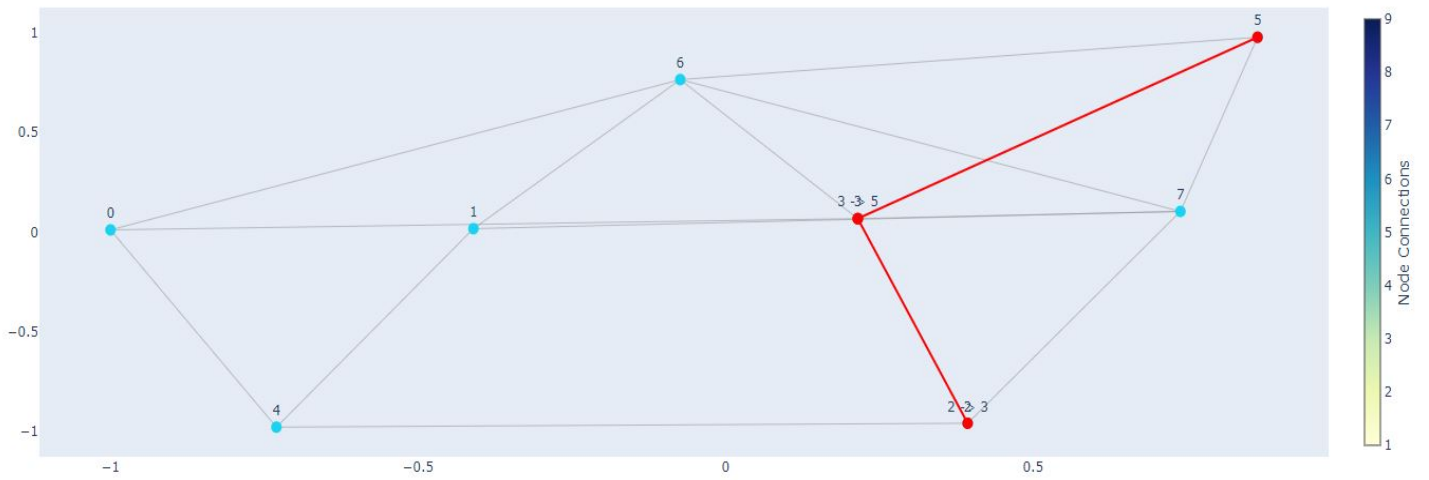


Figure: 5

Wireless Sensor Network - Step 4

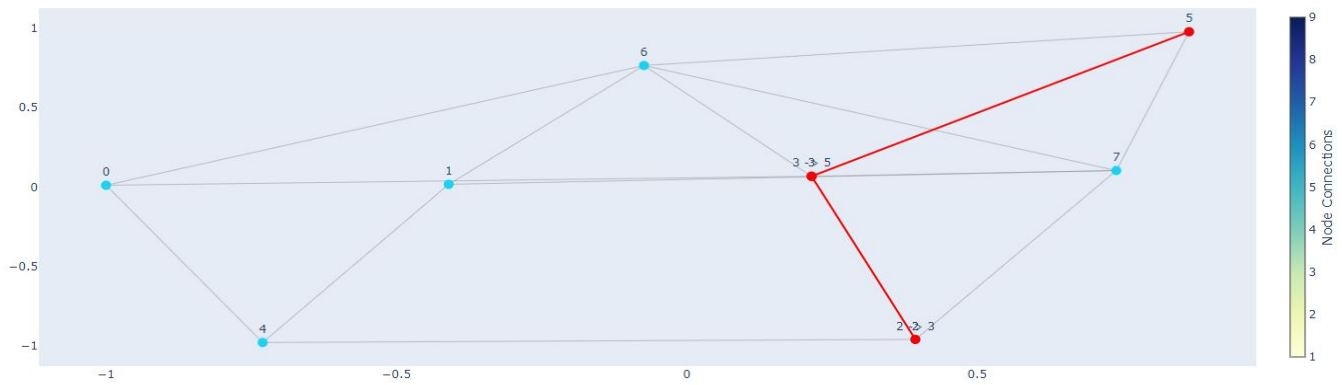


Figure : 6

Wireless Sensor Network - Step 6

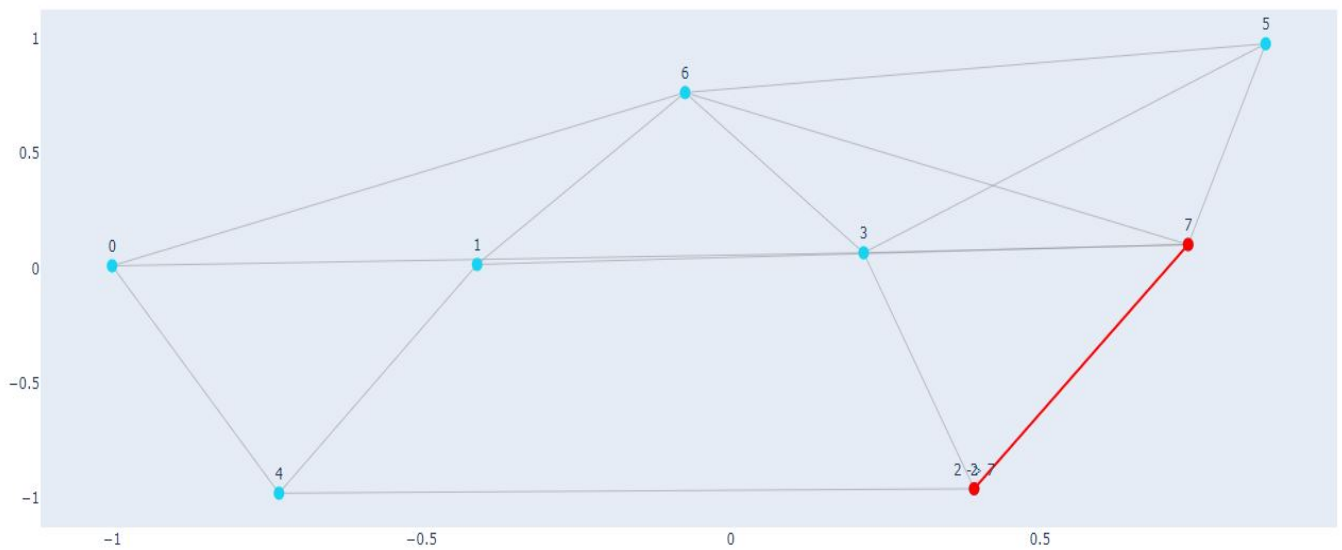


Figure: 7

Experiment 2 : Write a program Implement CSMA/CD MAC protocol in Wireless Sensor Network Using Python.

Explanation: (Step by step)

- **Graph Creation:**

- Creates a network graph `G` with 8 nodes and random edges. Uses `networkx` to generate node positions with a spring layout.

- **Dash Initialization:**

- Initializes a Dash web application with an empty layout containing a `dcc.Graph` for visualizing the network and an `dcc.Interval` component for periodic updates.

- **Transmission State Management:**

- Sets up a `transmission_state` dictionary to track simulation details like the current step, paths, timing, and node statuses (waiting, collision).

- **Create Plotly Figure:**

- Defines a function `create_figure` to generate a Plotly figure, including graph edges, nodes (with colors indicating status), and current communication paths.

- **Dash Callback:**

- Updates the network graph every second based on the simulation state, including handling different stages of the CSMA/CD protocol (carrier sensing, request, acknowledgment, data transfer).

- **CSMA/CD Protocol Simulation:**

- Implements stages of CSMA/CD protocol: carrier sensing, request, acknowledgment, and data transfer, with random backoff times on collision.

- **Path Tracking:**

- Uses a dictionary `path_busy` to track whether a path is currently in use and ensures only one path is active at a time.

- **Communication Simulation:**

- Simulates communication between nodes by generating paths and handling events like collisions and acknowledgments.

- **Function for Communication Simulation:**

- `simulate_communication` finds the shortest path between two nodes if it exists.

Code :

```
import networkx as nx
import plotly.graph_objects as go
import dash
from dash import dcc, html
from dash.dependencies import Input, Output
import random
from datetime import datetime
import threading
from collections import deque

# Create a graph for the wireless sensor network
G = nx.Graph()
num_nodes = 8
nodes = range(num_nodes)
G.add_nodes_from(nodes)
edges = [(i, j) for i in nodes for j in nodes if i < j and random.random() > 0.5]
G.add_edges_from(edges)
pos = nx.spring_layout(G)

# Initialize Dash app
app = dash.Dash(__name__)

# State to track the transmission
transmission_state = {
    'step': 0,
    'paths': [],
    'start_time': None,
    'stage': 'idle',
    'medium_free': True,
    'waiting_nodes': deque(),
    'collision': False,
    'backoff_time': 0,
    'active_path': None
}

# Dictionary to track the busy state of each path
path_busy = {}

# Lock for simulating medium access control
transmission_lock = threading.Lock()

# Function to generate Plotly figure
```

```

def create_figure(paths, step, current_time, waiting_nodes, collision):
    edge_trace = []
    for edge in G.edges():
        x0, y0 = pos[edge[0]]
        x1, y1 = pos[edge[1]]
        edge_trace.append(go.Scatter(
            x=[x0, x1, None], y=[y0, y1, None],
            line=dict(width=0.5, color='#888'),
            hoverinfo='none',
            mode='lines'))

    node_trace = go.Scatter(
        x=[],
        y=[],
        text=[],
        mode='markers+text',
        textposition='top center',
        hoverinfo='text',
        marker=dict(
            showscale=True,
            colorscale='YlGnBu',
            size=10,
            colorbar=dict(
                thickness=15,
                title='Node Connections',
                xanchor='left',
                titleside='right'
            ),
            color=[]
        )
    )

    for node in G.nodes():
        x, y = pos[node]
        node_trace['x'] += (x,)
        node_trace['y'] += (y,)
        if node in waiting_nodes:
            node_trace['text'] += (f'{node} (waiting)',)
            node_trace['marker']['color'] += ('orange',)
        else:
            node_trace['text'] += (str(node),)
            node_trace['marker']['color'] += ('blue',)

    path_trace = []
    for path in paths:
        if isinstance(path, list):
            elapsed_time = (current_time - transmission_state['start_time']).total_seconds()
            if elapsed_time < 50: # Show data transmission for 50 seconds

```

```

        for i in range(len(path) - 1):
            x0, y0 = pos[path[i]]
            x1, y1 = pos[path[i+1]]
            color = 'red' if collision else 'green'
            path_trace.append(go.Scatter(
                x=[x0, x1],
                y=[y0, y1],
                mode='markers+lines+text',
                marker=dict(size=10, color=color),
                line=dict(width=2, color=color),
                text=[f'{path[i]} -> {path[i+1]}'],
                textposition='top center'
            ))

fig = go.Figure(data=edge_trace + [node_trace] + path_trace,
                layout=go.Layout(
                    title=f'Wireless Sensor Network - Step {step}',
                    showlegend=False,
                    hovermode='closest',
                    margin=dict(b=20, l=5, r=5, t=40),
                    xaxis=dict(showgrid=False, zeroline=False),
                    yaxis=dict(showgrid=False, zeroline=False)))

return fig

# App layout
app.layout = html.Div([
    dcc.Graph(id='network-graph'),
    dcc.Interval(
        id='interval-component',
        interval=1*1000, # Update every second
        n_intervals=0
    )
])

@app.callback(
    Output('network-graph', 'figure'),
    [Input('interval-component', 'n_intervals')]
)
def update_graph(n):
    global transmission_state
    step = transmission_state['step']
    current_time = datetime.now()

    # Handle backoff state
    if transmission_state['stage'] == 'backoff':
        elapsed_time = (current_time - transmission_state['start_time']).total_seconds()
        if elapsed_time >= transmission_state['backoff_time']:

```



```

        print(f"Backoff time completed. Node {transmission_state['waiting_nodes'][0]} retrying
transmission.")
        transmission_state['collision'] = False
        transmission_state['stage'] = 'carrier_sensing'
        return create_figure(transmission_state['paths'], step, current_time,
transmission_state['waiting_nodes'], transmission_state['collision'])

# Only simulate communication steps up to 6 times
if step < 6:
    if transmission_state['start_time'] is None or (current_time -
transmission_state['start_time']).total_seconds() >= 50:
        if not transmission_state['waiting_nodes']:
            # Add new nodes to the waiting list if it's empty
            available_nodes = list(set(G.nodes()) - set(transmission_state['waiting_nodes']))
            if available_nodes:
                source = random.choice(available_nodes)
                target = random.choice(list(set(G.nodes()) - {source}))
                path = simulate_communication(G, source, target)
                if path:
                    transmission_state['paths'] = [path]
                    transmission_state['start_time'] = current_time
                    transmission_state['step'] += 1
                    transmission_state['stage'] = 'carrier_sensing'
                    transmission_state['waiting_nodes'].append(source)
            else:
                print("No available nodes to send data. Waiting...")
        else:
            print(f"Node {transmission_state['waiting_nodes'][0]} waiting to transmit.")

# CSMA/CD protocol stages
if transmission_state['stage'] == 'carrier_sensing':
    path = transmission_state['paths'][0]
    path_key = tuple(path)
    if path_key not in path_busy or not path_busy[path_key]:
        path_busy[path_key] = True
        print(f"Node {path[0]} detected that the channel is clear. Proceeding to request.")
        transmission_state['stage'] = 'request'
    else:
        print(f"Collision detected on path {path}. Node {path[0]} will backoff.")
        transmission_state['collision'] = True
        transmission_state['backoff_time'] = random.randint(1, 10)
        transmission_state['stage'] = 'backoff'

elif transmission_state['stage'] == 'request':
    path = transmission_state['paths'][0]
    with transmission_lock:
        path_key = tuple(path)
        if not path_busy[path_key]:

```

```

    path_busy[path_key] = True
    path_str = '-> '.join(map(str, path))
    print(f"Node {path[0]} is requesting communication with Node {path[-1]} via path: {path_str}")
    transmission_state['stage'] = 'acknowledgment'
else:
    print(f"Collision detected on path {path}.")
    transmission_state['collision'] = True
    transmission_state['backoff_time'] = random.randint(1, 10)
    transmission_state['stage'] = 'backoff'

elif transmission_state['stage'] == 'acknowledgment':
    elapsed_time = (current_time - transmission_state['start_time']).total_seconds()
    if elapsed_time >= 10: # Wait for 10 seconds to simulate acknowledgment
        path = transmission_state['paths'][0]
        path_str = '-> '.join(map(str, path))
        print(f"Receiver Node {path[-1]} has granted acknowledgment to Sender Node {path[0]} via path: {path_str}")
        transmission_state['stage'] = 'data_transfer'

elif transmission_state['stage'] == 'data_transfer':
    elapsed_time = (current_time - transmission_state['start_time']).total_seconds()
    if elapsed_time >= 20: # Wait for 20 seconds to simulate data transfer
        path = transmission_state['paths'][0]
        path_str = '-> '.join(map(str, path))
        print(f"Data transfer from Sender Node {path[0]} to Receiver Node {path[-1]} via path: {path_str}")
        print("Data transfer successful.")
        print("Releasing nodes and medium.")
        path_key = tuple(path)
        path_busy[path_key] = False
        transmission_state['waiting_nodes'].popleft()
        transmission_state['stage'] = 'idle'
        transmission_state['collision'] = False

    return create_figure(transmission_state['paths'], step, current_time,
transmission_state['waiting_nodes'], transmission_state['collision'])

# Function to simulate communication steps
def simulate_communication(G, source, target):
    if nx.has_path(G, source, target):
        path = nx.shortest_path(G, source, target)
        return path
    else:
        print(f"No path found from Node {source} to Node {target}")
        return []

if __name__ == '__main__':
    app.run_server(debug=True)

```

Output: (console printed)

Node 5 detected that the channel is clear. Proceeding to request.

Collision detected on path [5, 3].

Backoff time completed. Node 5 retrying transmission.

Collision detected on path [5, 3]. Node 5 will backoff.

Backoff time completed. Node 5 retrying transmission.

Node 5 waiting to transmit.

Collision detected on path [5, 3]. Node 5 will backoff.

Backoff time completed. Node 5 retrying transmission.

Node 5 waiting to transmit.

Visualization Output-

Figure8: We can see detect the collision on the path representing through green color.

Wireless Sensor Network - Step 0

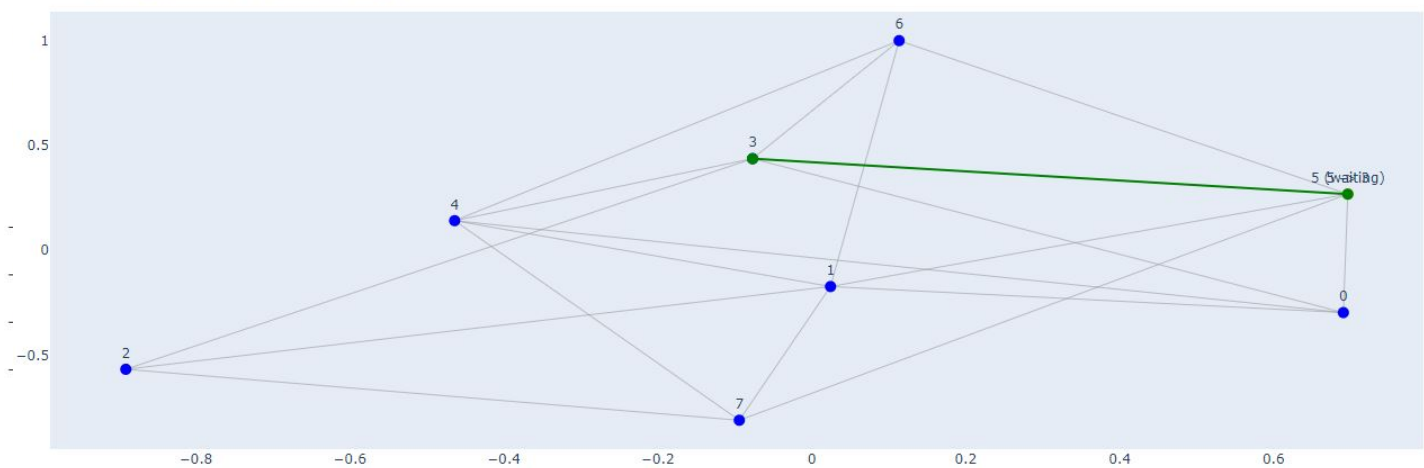
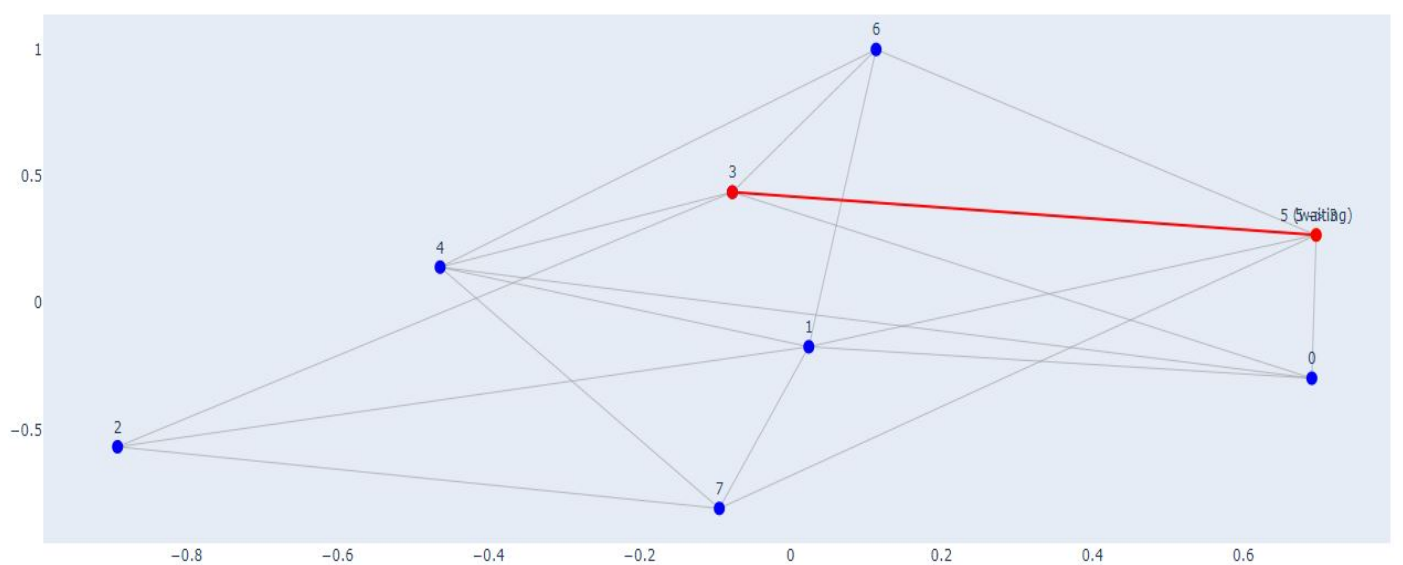


Figure9: We can see after collision free on the path transmit data representing through red color.

Wireless Sensor Network - Step 1



Experiment 3: Write a program Implement Schedule Based MAC protocol like TDMA , FDMA In Wireless Communication Using Python.

Explanation: A scheduled-based MAC (Medium Access Control) protocol is a type of network protocol used to manage access to the communication medium in a network by assigning specific times or frequencies for transmission. This approach helps in preventing collisions and optimizing the use of network resources. Scheduled-based MAC protocols are commonly used in networks where predictable and efficient communication is crucial, such as in wireless sensor networks (WSNs). I can explain FDMA and TDMA used to code by implementing scheduled based mac protocol.

- **TDMA (Time Division Multiple Access):**

- **Functionality:** TDMA divides the time into slots, and each node or communication session is allocated a specific time slot. Nodes can only transmit data during their assigned slot.
- **In Code:**
 - **Slot Management:** The `transmission_state` manages the current TDMA slot and ensures that communication is scheduled according to the slot.
 - **Communication Simulation:** In the `update_graph` function, the `simulate_tdma_communication` function is used to determine the communication path between nodes for each TDMA slot.

- **FDMA (Frequency Division Multiple Access):**

- **Functionality:** FDMA divides the available frequency spectrum into distinct frequency bands, each of which can be used by different nodes or communication sessions.
- **In Code:**
 - **Frequency Management:** The `transmission_state` maintains a list of frequency bands and assigns them based on the current slot.
 - **Communication Simulation:** The `simulate_fdma_communication` function determines the communication path between nodes, and the frequency band is printed as part of the output.

CODE :

```
import networkx as nx
import plotly.graph_objects as go
import dash
from dash import dcc, html
from dash.dependencies import Input, Output
import random
from datetime import datetime, timedelta
import logging

# Configure logging
```

```

logging.basicConfig(level=logging.DEBUG)

# Create a graph for the wireless sensor network
G = nx.Graph()

# Number of nodes
num_nodes = 8

# Add nodes to the graph
nodes = range(num_nodes)
G.add_nodes_from(nodes)

# Randomly add edges between nodes to simulate wireless connections
edges = [(i, j) for i in nodes for j in nodes if i < j and random.random() > 0.5]
G.add_edges_from(edges)

# Assign positions to nodes for visualization
pos = nx.spring_layout(G)

# Initialize Dash app
app = dash.Dash(__name__)

# State to track the transmission
transmission_state = {
    'step': 0,
    'paths': [],
    'start_time': None,
    'current_slot': 0,
    'slot_duration': 50, # Duration of each TDMA slot in seconds
    'frequency_bands': ['2.4GHz', '2.5GHz'],
    'last_update': datetime.now()
}

# Function to generate Plotly figure
def create_figure(paths, step, current_time):
    edge_trace = []
    for edge in G.edges():
        x0, y0 = pos[edge[0]]
        x1, y1 = pos[edge[1]]
        edge_trace.append(go.Scatter(
            x=[x0, x1, None], y=[y0, y1, None],
            line=dict(width=0.5, color='#888'),
            hoverinfo='none',
            mode='lines'))

    node_trace = go.Scatter(
        x=[], y=[],
        text=[],

```

```

mode='markers+text',
textposition='top center',
hoverinfo='text',
marker=dict(
    showscale=True,
    colorscale='YlGnBu',
    size=10,
    colorbar=dict(
        thickness=15,
        title='Node Connections',
        xanchor='left',
        titleside='right'
    )
)
)

for node in G.nodes():
    x, y = pos[node]
    node_trace['x'] += tuple([x])
    node_trace['y'] += tuple([y])
    node_trace['text'] += tuple([str(node)])

# Create a figure
fig = go.Figure(data=edge_trace + [node_trace],
    layout=go.Layout(
        title=f'Wireless Sensor Network - Step {step}',
        showlegend=False,
        hovermode='closest',
        margin=dict(b=20, l=5, r=5, t=40),
        xaxis=dict(showgrid=False, zeroline=False),
        yaxis=dict(showgrid=False, zeroline=False)))

# Add arrows for communication paths
for path in paths:
    if isinstance(path, list):
        elapsed_time = (current_time - transmission_state['start_time']).total_seconds()
        if elapsed_time < 50:
            for i in range(len(path) - 1):
                x0, y0 = pos[path[i]]
                x1, y1 = pos[path[i+1]]
                fig.add_trace(go.Scatter(
                    x=[x0, x1],
                    y=[y0, y1],
                    mode='markers+lines+text',
                    marker=dict(size=10, color='red'),
                    line=dict(width=2, color='red'),
                    text=[f'{path[i]} -> {path[i+1]}'],
                    textposition='top center'
                ))

```

```

        ))

    return fig

# App layout
app.layout = html.Div([
    dcc.Graph(id='network-graph'),
    dcc.Interval(
        id='interval-component',
        interval=1*1000, # Update every second
        n_intervals=0
    )
])

@app.callback(
    Output('network-graph', 'figure'),
    [Input('interval-component', 'n_intervals')]
)
def update_graph(n):
    global transmission_state

    # Update the current time
    current_time = datetime.now()

    # Check if it's time to switch TDMA slot
    if transmission_state['start_time'] is None:
        transmission_state['start_time'] = current_time

    elapsed_time = (current_time - transmission_state['start_time']).total_seconds()
    if elapsed_time >= transmission_state['slot_duration']:
        transmission_state['step'] += 1
        transmission_state['start_time'] = current_time
        transmission_state['current_slot'] += 1
        if transmission_state['step'] % 2 == 0:
            # Simulate TDMA communication
            source, target = random.sample(nodes, 2)
            path = simulate_tdma_communication(G, source, target)
        else:
            # Simulate FDMA communication
            source, target = random.sample(nodes, 2)
            path = simulate_fdma_communication(G, source, target)

        if path:
            transmission_state['paths'] = [path]
            fdma_band = transmission_state['frequency_bands'][transmission_state['current_slot'] %
len(transmission_state['frequency_bands'])]
            if transmission_state['step'] % 2 == 0:
                # Print TDMA communication details

```

```

        print(f"TDMA: Communication between Node {path[0]} and Node {path[-1]} using path: {' -> '.join(map(str, path))}")
        print(f"TDMA Slot: {transmission_state['current_slot']}")
        print(f"Sender Node {path[0]} is requesting communication with Receiver Node {path[-1]} via path: {' -> '.join(map(str, path))}")
    else:
        # Print FDMA communication details
        print(f"FDMA: Communication between Node {path[0]} and Node {path[-1]} using path: {' -> '.join(map(str, path))}")
        print(f"FDMA Frequency Band: {fdma_band}")
        print(f"Sender Node {path[0]} is requesting communication with Receiver Node {path[-1]} via path: {' -> '.join(map(str, path))}")

    return create_figure(transmission_state['paths'], transmission_state['step'], current_time)

# Function to simulate TDMA communication steps
def simulate_tdma_communication(G, source, target):
    if nx.has_path(G, source, target):
        path = nx.shortest_path(G, source, target)
        return path
    else:
        print(f"No path found from Node {source} to Node {target}")
        return []

# Function to simulate FDMA communication steps
def simulate_fdma_communication(G, source, target):
    if nx.has_path(G, source, target):
        path = nx.shortest_path(G, source, target)
        return path
    else:
        print(f"No path found from Node {source} to Node {target}")
        return []

if __name__ == '__main__':
    app.run_server(debug=True)

```

Output: (console printed)

Figure 11

```

FDMA Frequency Band: 2.5GHz
Sender Node 4 is requesting communication with Receiver Node 6 via path: 4 -> 2 -> 6
Now Node 6 Acknowledge for sending data ,
[node 4 -> node 6] Transmit data.....

```

Figure 12

```

TDMA: Communication between Node 6 and Node 0 using path: 6 -> 2 -> 0
TDMA Slot: 2
Sender Node 6 is requesting communication with Receiver Node 0 via path: 6 -> 2 -> 0
Now Node 0 Acknowledge for sending data ,
[node 6 -> node 0] Transmit data.....

```


Visualized Output :

Wireless Sensor Network - Step 0

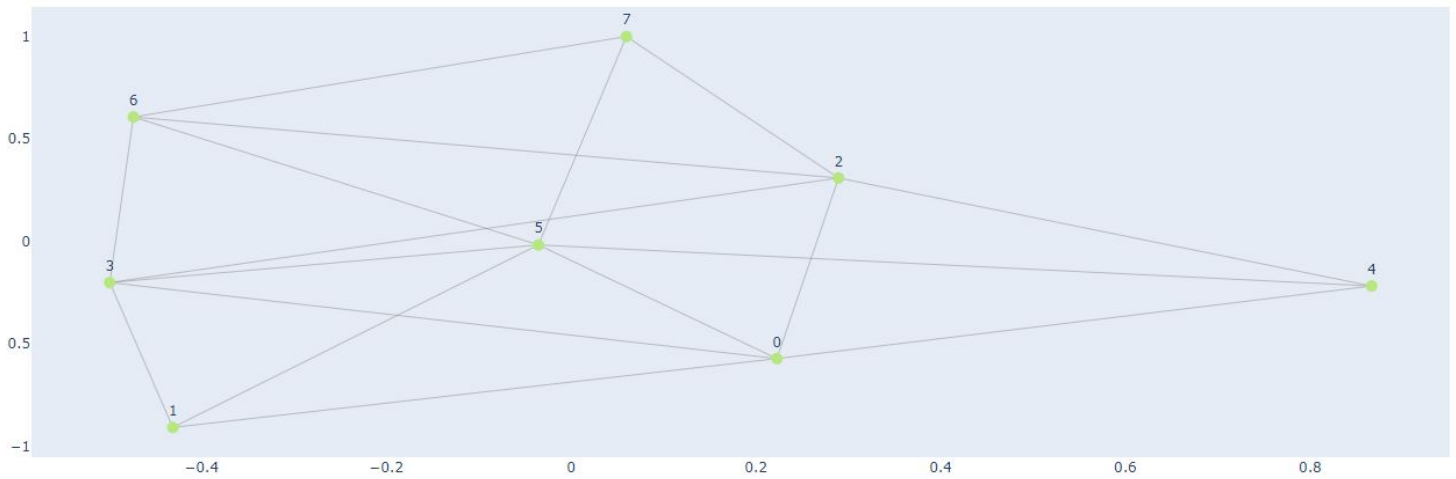


Figure : 10 (Initial Network)

Wireless Sensor Network - Step 1

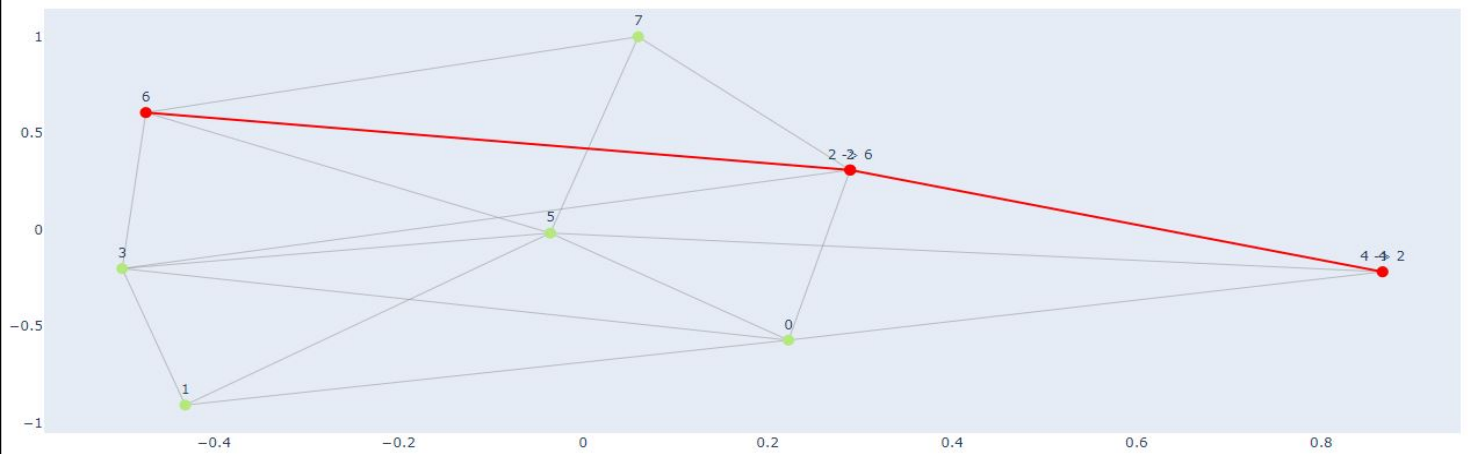


Figure : 11 (FDMA Scheduled Mac Protocol Network)

Wireless Sensor Network - Step 2

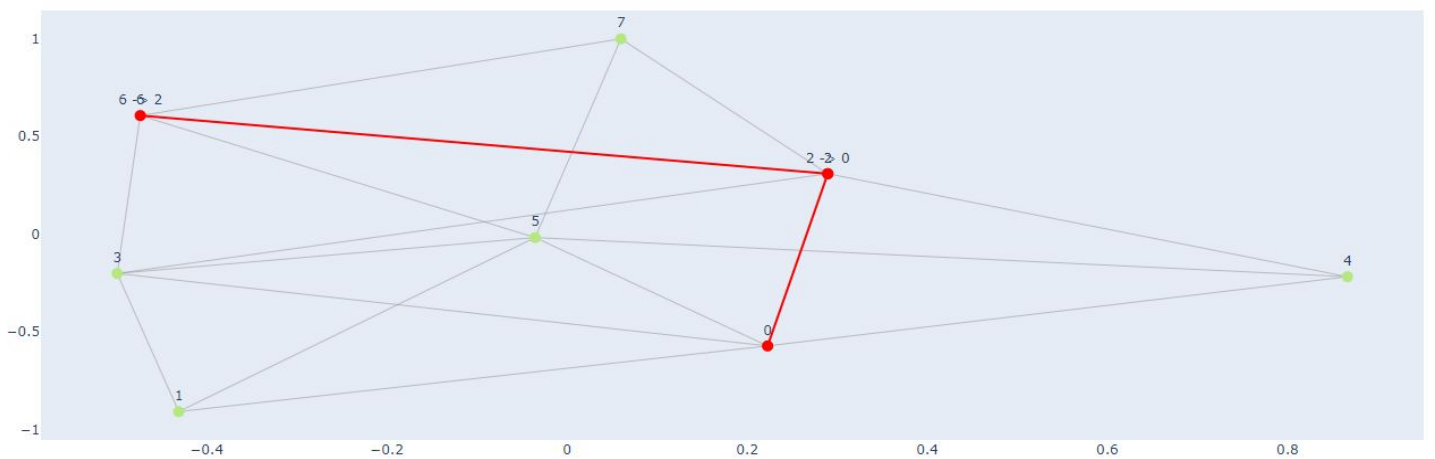


Figure : 12 (TDMA Scheduled Mac Protocol Network)

Experiment 4 : Simulating and Testing Routing Protocols in Wireless Sensor Networks (WSN).

Explanation : The Wireless Sensor Network (WSN) simulation experiment is designed to model and simulate the behavior of wireless sensor networks, which are commonly used in various applications such as environmental monitoring, smart agriculture, and industrial automation. This experiment focuses on simulating data transmission using different routing protocols (AODV and DSR) and network topologies (grid, random, and cluster).

Key Components

1. **Node Class**: Represents individual nodes in the network, each of which can function as a sensor or a base station. Nodes can queue data, process data, and forward data to other nodes.
2. **Network Class**: Manages the collection of nodes and the connections (links) between them. It supports generating different network topologies and running simulations using specified routing protocols.
3. **Command-Line Interface (CLI)**: Allows users to configure and run simulations from the command line, specifying parameters such as the routing protocol, number of nodes, number of simulation steps, and the network topology.
4. **Unit Tests**: Ensure the correctness of the implemented functionalities, testing node creation, adding nodes to the network, establishing links, and running simulations.

Code :

Node.py –

```
# wsn_sim/node.py
import random
```

```
class Node:
```

```
    def __init__(self, node_id, position, role='sensor'):
        self.node_id = node_id
        self.position = position
        self.role = role
        self.energy = 100 # Assume all nodes start with 100 units of energy
        self.data_queue = []
```

```
    def queue_data(self, data):
        self.data_queue.append(data)
```

```
    def process_data_queue(self, network):
        while self.data_queue:
            data = self.data_queue.pop(0)
            if data['destination'] == self:
                print(f"Node {self.node_id} received data: {data['content']}")
```

```

    else:
        recipient = self.find_next_hop_aodv(network, data['destination'])
        if recipient:
            print(f"Node {self.node_id} forwarding data to Node {recipient.node_id}")
            self.send_data(recipient, data)

def find_next_hop_aodv(self, network, destination):
    # Simplified routing: forward to a random neighbor (for demo purposes)
    neighbors = network.get_neighbors(self)
    if not neighbors:
        return None
    return random.choice(neighbors)

def send_data(self, recipient, data):
    recipient.queue_data(data)

def send_rreq(self, network, destination):
    print(f"Node {self.node_id} broadcasting RREQ for Node {destination.node_id}")
    rreq = {'source': self, 'destination': destination}
    for neighbor in network.get_neighbors(self):
        network.send_rreq(neighbor, rreq)

def receive_rreq(self, network, rreq):
    if rreq['destination'] == self:
        print(f"Node {self.node_id} received RREQ from Node {rreq['source'].node_id}")
        rrep = {'source': self, 'destination': rreq['source']}
        network.send_rrep(rreq['source'], rrep)
    else:
        self.send_rreq(network, rreq['destination'])

def receive_rrep(self, network, rrep):
    print(f"Node {self.node_id} received RREP from Node {rrep['source'].node_id}")

def send_route_request(self, network, route_request):
    print(f"Node {self.node_id} broadcasting route request")
    for neighbor in network.get_neighbors(self):
        network.send_route_request(neighbor, route_request)

def receive_route_request(self, network, route_request):
    print(f"Node {self.node_id} received route request from Node {route_request['source'].node_id}")

def receive_route_reply(self, network, route_reply):
    print(f"Node {self.node_id} received route reply from Node {route_reply['source'].node_id}")

```

Network.py –
 # wsn_sim/network.py
 import random
 import networkx as nx

```

import numpy as np
from node import Node
import matplotlib.pyplot as plt

class Network:
    def __init__(self):
        self.nodes = []
        self.graph = nx.Graph()
        self.node_mapping = {}

    def add_node(self, node):
        self.nodes.append(node)
        self.node_mapping[node.node_id] = node
        self.graph.add_node(node.node_id, pos=node.position)

    def add_link(self, node1_id, node2_id):
        self.graph.add_edge(node1_id, node2_id)

    def generate_topology(self, topology_type, nodes, links):
        if topology_type == 'grid':
            self.generate_grid_topology(nodes)
        elif topology_type == 'random':
            self.generate_random_topology(nodes, links)
        elif topology_type == 'cluster':
            self.generate_cluster_topology(nodes, links)

    def generate_grid_topology(self, nodes):
        grid_size = int(np.ceil(np.sqrt(nodes)))
        for i in range(nodes):
            x, y = divmod(i, grid_size)
            node = Node(i, (x, y))
            self.add_node(node)
        for i in range(grid_size):
            for j in range(grid_size):
                if j < grid_size - 1:
                    self.add_link(i * grid_size + j, i * grid_size + j + 1)
                if i < grid_size - 1:
                    self.add_link(i * grid_size + j, (i + 1) * grid_size + j)

    def generate_random_topology(self, nodes, links):
        for i in range(nodes):
            node = Node(i, (random.randint(0, 100), random.randint(0, 100)))
            self.add_node(node)
        for _ in range(links):
            node1, node2 = random.sample(self.nodes, 2)
            self.add_link(node1.node_id, node2.node_id)

    def generate_cluster_topology(self, nodes, links):

```

```

clusters = int(np.sqrt(nodes))
nodes_per_cluster = nodes // clusters
cluster_centers = [(random.randint(0, 100), random.randint(0, 100)) for _ in range(clusters)]
for cluster_id, center in enumerate(cluster_centers):
    for i in range(nodes_per_cluster):
        position = (center[0] + random.randint(-10, 10), center[1] + random.randint(-10, 10))
        node = Node(cluster_id * nodes_per_cluster + i, position)
        self.add_node(node)
for _ in range(links):
    node1, node2 = random.sample(self.nodes, 2)
    self.add_link(node1.node_id, node2.node_id)

def run_aodv_simulation(self, steps):
    for step in range(steps):
        print(f"Simulation step {step + 1}")
        for node in self.nodes:
            if node.role == 'sensor' and node.energy > 0:
                data = f"Temperature: {random.uniform(15, 35):.2f}C"
                node.queue_data({'destination': self.get_base_station(), 'content': data})
                node.process_data_queue(self)

def run_dsr_simulation(self, steps):
    for step in range(steps):
        print(f"Simulation step {step + 1}")
        for node in self.nodes:
            if node.role == 'sensor' and node.energy > 0:
                data = f"Temperature: {random.uniform(15, 35):.2f}C"
                node.queue_data({'destination': self.get_base_station(), 'content': data})
                node.process_data_queue(self)

def get_base_station(self):
    return next(node for node in self.nodes if node.role == 'base_station')

def get_neighbors(self, node):
    neighbors_ids = list(self.graph.neighbors(node.node_id))
    return [self.node_mapping[n_id] for n_id in neighbors_ids]

def send_rreq(self, neighbor, rreq):
    neighbor.receive_rreq(self, rreq)

def send_rrep(self, neighbor, rrep):
    neighbor.receive_rrep(self, rrep)

def send_route_request(self, neighbor, route_request):
    neighbor.receive_route_request(self, route_request)

def send_route_reply(self, neighbor, route_reply):
    neighbor.receive_route_reply(self, route_reply)

```

```

def visualize(self, filename='graph_visualization.png'):
    pos = nx.get_node_attributes(self.graph, 'pos')
    energy_levels = [node.energy for node in self.nodes]
    plt.figure(figsize=(10, 8))
    if len(energy_levels) == len(self.graph.nodes):
        nx.draw(self.graph, pos, with_labels=True, node_color='skyblue', node_size=[e * 10 for e in
energy_levels])
    else:
        nx.draw(self.graph, pos, with_labels=True, node_color='skyblue', node_size=100)
    plt.title('Wireless Sensor Network Visualization')
    plt.savefig(filename)
    plt.close()
    print(f"Graph saved to {filename}")

```

Cli.py –

```

# wsn_sim/cli.py
import click
import configparser
from network import Network
from node import Node

def read_config(file_path):
    config = configparser.ConfigParser()
    config.read(file_path)
    simulation_config = {
        'protocol': config.get('simulation', 'protocol'),
        'steps': config.getint('simulation', 'steps'),
        'nodes': config.getint('simulation', 'nodes'),
        'links': config.getint('simulation', 'links'),
        'topology': config.get('simulation', 'topology')
    }
    return simulation_config

@click.command()
@click.option('--config', type=click.Path(), help='Path to the configuration file')
@click.option('--protocol', type=str, default='AODV', help='Routing protocol (AODV/DSR)')
@click.option('--steps', type=int, default=5, help='Number of simulation steps')
@click.option('--nodes', type=int, default=10, help='Number of nodes in the network')
@click.option('--links', type=int, default=15, help='Number of random links between nodes')
@click.option('--topology', type=str, default='cluster', help='Network topology (grid/random/cluster)')
def run_simulation(config, protocol, steps, nodes, links, topology):
    if config:
        config_values = read_config(config)
        protocol = config_values['protocol']
        steps = config_values['steps']
        nodes = config_values['nodes']
        links = config_values['links']

```

```

    topology = config_values['topology']

    net = Network()
    net.generate_topology(topology, nodes, links)

    base_station = Node(0, (50, 50), role='base_station')
    net.add_node(base_station)

    if protocol.upper() == 'AODV':
        net.run_aodv_simulation(steps)
    elif protocol.upper() == 'DSR':
        net.run_dsr_simulation(steps)

    net.visualize()

if __name__ == '__main__':
    run_simulation()

```

Now Simulate by using the command : ***python wsn_sim/cli.py --protocol AODV --steps 5 --nodes 5 --links 5 --topology cluster***

Output : (console printed)-

Simulation step 1

Node 0 forwarding data to Node 3
 Node 1 forwarding data to Node 3
 Node 2 forwarding data to Node 0
 Node 3 forwarding data to Node 1
 Node 3 forwarding data to Node 0
 Node 3 forwarding data to Node 1
 Node 0 received data: Temperature: 32.56C
 Node 0 received data: Temperature: 27.76C

Simulation step 2

Node 0 forwarding data to Node 2
 Node 1 forwarding data to Node 3
 Node 1 forwarding data to Node 3
 Node 1 forwarding data to Node 3
 Node 2 forwarding data to Node 0
 Node 2 forwarding data to Node 0
 Node 3 forwarding data to Node 1
 Node 3 forwarding data to Node 0
 Node 3 forwarding data to Node 1
 Node 3 forwarding data to Node 1
 Node 0 received data: Temperature: 19.10C
 Node 0 received data: Temperature: 28.93C
 Node 0 received data: Temperature: 21.02C

Simulation step 3

Node 0 forwarding data to Node 3
Node 1 forwarding data to Node 3
Node 1 forwarding data to Node 3
Node 1 forwarding data to Node 3
Node 1 forwarding data to Node 3
Node 2 forwarding data to Node 0
Node 3 forwarding data to Node 1
Node 3 forwarding data to Node 1
Node 3 forwarding data to Node 1
Node 3 forwarding data to Node 1
Node 3 forwarding data to Node 0
Node 3 forwarding data to Node 0
Node 0 received data: Temperature: 15.57C
Node 0 received data: Temperature: 28.07C
Node 0 received data: Temperature: 25.85C
Simulation step 4
Node 0 forwarding data to Node 3
Node 1 forwarding data to Node 3
Node 1 forwarding data to Node 3
Node 1 forwarding data to Node 3
Node 1 forwarding data to Node 3
Node 1 forwarding data to Node 3
Node 2 forwarding data to Node 0
Node 3 forwarding data to Node 0
Node 3 forwarding data to Node 0
Node 3 forwarding data to Node 1
Node 3 forwarding data to Node 1
Node 3 forwarding data to Node 1
Node 3 forwarding data to Node 0
Node 3 forwarding data to Node 1
Node 0 received data: Temperature: 17.16C
Node 0 received data: Temperature: 19.85C
Node 0 received data: Temperature: 24.11C
Node 0 received data: Temperature: 19.67C
Simulation step 5
Node 0 forwarding data to Node 3
Node 1 forwarding data to Node 3
Node 1 forwarding data to Node 3
Node 1 forwarding data to Node 3
Node 1 forwarding data to Node 3
Node 1 forwarding data to Node 3
Node 2 forwarding data to Node 0
Node 3 forwarding data to Node 0
Node 3 forwarding data to Node 0

Node 3 forwarding data to Node 0
Node 3 forwarding data to Node 0
Node 3 forwarding data to Node 1
Node 3 forwarding data to Node 1
Node 3 forwarding data to Node 0
Node 0 received data: Temperature: 24.97C
Node 0 received data: Temperature: 34.36C
Node 0 received data: Temperature: 16.19C
Node 0 received data: Temperature: 31.19C
Node 0 received data: Temperature: 25.94C
Node 0 received data: Temperature: 26.49C
Graph saved to graph_visualization.png

Visualize output :

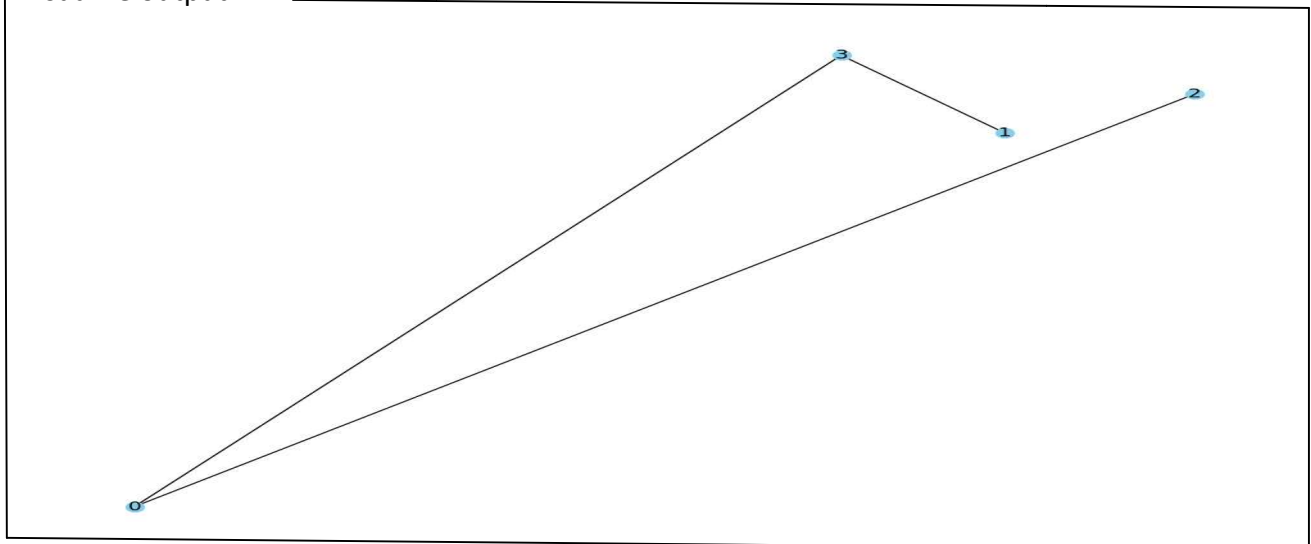


Figure : 13

For Testing run the code:

```
# tests/test_network.py

import unittest

from wsn_sim.node import Node

from wsn_sim.network import Network

import random

class TestWirelessSensorNetwork(unittest.TestCase):
```

```

def setUp(self):

    self.net = Network()

    self.base_station = Node(0, (50, 50), role='base_station')

    self.net.add_node(self.base_station)


    for i in range(1, 5):

        node = Node(i, (random.randint(0, 100), random.randint(0, 100)))

        self.net.add_node(node)


    self.net.add_link(0, 1)

    self.net.add_link(1, 2)

    self.net.add_link(2, 3)

    self.net.add_link(3, 4)


def test_node_creation(self):

    node = Node(5, (10, 10))

    self.assertEqual(node.node_id, 5)

    self.assertEqual(node.position, (10, 10))

    self.assertEqual(node.role, 'sensor')

    self.assertEqual(node.energy, 100)


def test_add_node_to_network(self):

    node = Node(6, (20, 20))

    self.net.add_node(node)

```

```

        self.assertIn(node, self.net.nodes)

def test_add_link(self):

    self.net.add_link(0, 2)

    self.assertTrue(self.net.graph.has_edge(0, 2))

def test_node_transmit(self):

    node1 = self.net.node_mapping[1]

    node2 = self.net.node_mapping[2]

    initial_energy = node1.energy

    data = {'content': 'Hello', 'destination': node2}

    node1.send_data(node2, data)

    self.assertLess(node1.energy, initial_energy)

def test_run_simulation(self):

    self.net.run_aodv_simulation(steps=2)

    for node in self.net.nodes:

        self.assertIsNotNone(node.energy)

if __name__ == '__main__':

    unittest.main()

```

Simulate vy using the command : ***python -m unittest discover -s tests***

Output :

Used AODV protocol -

.Simulation step 1

Node 1 broadcasting Route Request for Node 0

Node 0 sending Route Reply to Node 1

Node 2 broadcasting Route Request for Node 0
 Node 0 sending Route Reply to Node 2
 Node 3 broadcasting Route Request for Node 0
 Node 0 sending Route Reply to Node 3
 Node 4 broadcasting Route Request for Node 0
 Node 0 sending Route Reply to Node 4
 Simulation step 2
 Node 1 transmitted data to Node 3. Remaining energy: 97.1721530298589
 Node 3 received data. Remaining energy: 99.05
 Node 2 transmitted data to Node 3. Remaining energy: 97.78855176995205
 Node 3 received data. Remaining energy: 98.1
 Node 3 transmitted data to Node 3. Remaining energy: 96.19999999999999
 Node 3 received data. Remaining energy: 95.24999999999999
 Node 4 transmitted data to Node 3. Remaining energy: 97.59840255184062
 Node 3 received data. Remaining energy: 94.29999999999998
Node 1 transmitted data to Node 2. Remaining energy: 99.07573593128807
 Used DSR protocol -
 .Simulation step 1
 Node 1 broadcasting RREQ for Node 0
 Node 0 sending RREP to Node 1
 Node 2 broadcasting RREQ for Node 0
 Node 0 sending RREP to Node 2
 Node 3 broadcasting RREQ for Node 0
 Node 0 sending RREP to Node 3
 Node 4 broadcasting RREQ for Node 0
 Node 0 sending RREP to Node 4
 Simulation step 2
 Node 1 transmitted data to Node 0. Remaining energy: 97.68226802851589
 Node 0 received data. Remaining energy: 99.05
 Node 2 transmitted data to Node 0. Remaining energy: 97.87174575578973
 Node 0 received data. Remaining energy: 98.1
 Node 3 transmitted data to Node 0. Remaining energy: 97.7172468158199
 Node 0 received data. Remaining energy: 97.14999999999999
 Node 4 transmitted data to Node 0. Remaining energy: 97.53697246958963
 Node 0 received data. Remaining energy: 96.19999999999999
 .

 Run 10 tests in 0.009s

Experiment 5: Simulating and Testing Leach Routing Protocols in Wireless Sensor Networks (WSN).

Explanation : This code simulates a wireless sensor network (WSN) using the LEACH (Low-Energy Adaptive Clustering Hierarchy) protocol. Here's a brief explanation:

1. **Node Class:** Represents a node with attributes like ID, position, energy, state, and cluster head status. Nodes can be in 'ACTIVE' or 'SLEEP' state and can compute energy consumption based on their state.
2. **Network Class:** Manages a collection of nodes, initializes them with random positions, and tracks alive nodes.
3. **LEACH Setup Phase:**
 - o **Cluster Head Selection:** Nodes are probabilistically chosen as cluster heads based on a ratio ($\text{NB_CLUSTERS} / \text{NB_NODES}$). Cluster heads are assigned a next hop to the base station.
 - o **Ordinary Node Assignment:** Non-cluster head nodes find and set the nearest cluster head as their next hop. Cluster heads and ordinary nodes are set to 'ACTIVE' and 'SLEEP', respectively.
4. **Plot Network:** Visualizes the network using NetworkX and Matplotlib, showing cluster heads and their connections to other nodes.
5. **Main Execution:** Runs the LEACH setup phase and plots the network for a number of simulation rounds.

The LEACH protocol helps in energy-efficient data gathering by reducing the communication load on nodes through clustering.

CODE :

```
import numpy as np
import matplotlib.pyplot as plt
import networkx as nx

# Define constants
NB_NODES = 20 # Total number of nodes
NB_CLUSTERS = 5 # Number of clusters
SIMULATION_ROUNDS = 6 # Number of rounds
DATA_AGGREGATION_COST = 0.1 # Cost of data aggregation (arbitrary units)
SLEEP_MODE_COST = 0.05 # Cost of being in sleep mode (arbitrary units)
ACTIVE_MODE_COST = 0.2 # Cost of being in active mode (arbitrary units)

class Node:
    def __init__(self, node_id, x, y):
        self.node_id = node_id
        self.x = x
        self.y = y
        self.is_cluster_head = False
        self.next_hop = None
```

```

    self.energy = 1.0 # Initial energy
    self.state = 'ACTIVE' # Can be 'ACTIVE' or 'SLEEP'

def distance_to(self, other_node):
    return np.sqrt((self.x - other_node.x) ** 2 + (self.y - other_node.y) ** 2)

def consume_energy(self, mode):
    if mode == 'ACTIVE':
        self.energy -= ACTIVE_MODE_COST
    elif mode == 'SLEEP':
        self.energy -= SLEEP_MODE_COST

def __str__(self):
    return f"Node {self.node_id}, Energy: {self.energy:.2f}, State: {self.state}"

class Network:
    def __init__(self):
        self.nodes = []
        self.create_nodes()

    def create_nodes(self):
        for i in range(NB_NODES):
            x = np.random.uniform(0, 100)
            y = np.random.uniform(0, 100)
            self.nodes.append(Node(i, x, y))

    def get_alive_nodes(self):
        return [node for node in self.nodes if node.energy > 0]

    def broadcast_next_hop(self):
        for node in self.nodes:
            print(f"Node {node.node_id} -> Next Hop: {node.next_hop}")
            print(node)

    def data_aggregation_cost(self, node):
        return DATA_AGGREGATION_COST if node.is_cluster_head else 0

def leach_setup_phase(network):
    prob_ch = NB_CLUSTERS / NB_NODES
    heads = []
    alive_nodes = network.get_alive_nodes()

    # Decide which nodes are cluster heads
    idx = 0
    while len(heads) < NB_CLUSTERS:
        node = alive_nodes[idx]
        if np.random.uniform(0, 1) < prob_ch:
            node.is_cluster_head = True

```

```

        node.next_hop = 'BSID' # Base Station ID
        heads.append(node)
        idx = (idx + 1) % len(alive_nodes)

# Ordinary nodes choose nearest cluster heads
for node in alive_nodes:
    if node.is_cluster_head:
        node.consume_energy('ACTIVE') # CH is active
        node.state = 'ACTIVE'
    else:
        nearest_head = heads[0]
        for head in heads[1:]:
            if node.distance_to(head) < node.distance_to(nearest_head):
                nearest_head = head
        node.next_hop = nearest_head.node_id
        node.consume_energy('SLEEP') # Ordinary nodes sleep
        node.state = 'SLEEP'

network.broadcast_next_hop()

def plot_network(nodes):
    G = nx.Graph()
    pos = {node.node_id: (node.x, node.y) for node in nodes}

    # Add nodes to the graph
    for node in nodes:
        G.add_node(node.node_id, pos=(node.x, node.y), color='blue' if node.is_cluster_head else 'red')

    # Add edges based on next_hop
    for node in nodes:
        if node.next_hop != 'BSID' and node.next_hop in G.nodes:
            G.add_edge(node.node_id, node.next_hop)

    # Draw the network
    colors = [G.nodes[n]['color'] for n in G.nodes]
    nx.draw(G, pos, node_color=colors, with_labels=True, node_size=100, edge_color='gray', font_size=8,
font_color='black')

plt.title('Network Nodes with LEACH Protocol')
plt.show()

# Main execution
network = Network()
for _ in range(SIMULATION_ROUNDS):
    leach_setup_phase(network)
    plot_network(network.get_alive_nodes())

```

Output :

Node 0 -> Next Hop: BSID
Node 0, Energy: 0.80, State: ACTIVE
Node 1 -> Next Hop: 13
Node 1, Energy: 0.95, State: SLEEP
Node 2 -> Next Hop: 13
Node 2, Energy: 0.95, State: SLEEP
Node 3 -> Next Hop: 12
Node 3, Energy: 0.95, State: SLEEP
Node 4 -> Next Hop: 0
Node 4, Energy: 0.95, State: SLEEP
Node 5 -> Next Hop: BSID
Node 5, Energy: 0.80, State: ACTIVE
Node 6 -> Next Hop: 0
Node 6, Energy: 0.95, State: SLEEP
Node 7 -> Next Hop: 12
Node 7, Energy: 0.95, State: SLEEP
Node 8 -> Next Hop: BSID
Node 8, Energy: 0.80, State: ACTIVE
Node 9 -> Next Hop: 12
Node 9, Energy: 0.95, State: SLEEP
Node 10 -> Next Hop: 8
Node 10, Energy: 0.95, State: SLEEP
Node 11 -> Next Hop: 5
Node 11, Energy: 0.95, State: SLEEP
Node 12 -> Next Hop: BSID
Node 12, Energy: 0.80, State: ACTIVE
Node 13 -> Next Hop: BSID
Node 13, Energy: 0.80, State: ACTIVE
Node 14 -> Next Hop: 8
Node 14, Energy: 0.95, State: SLEEP
Node 15 -> Next Hop: 13
Node 15, Energy: 0.95, State: SLEEP
Node 16 -> Next Hop: 12
Node 16, Energy: 0.95, State: SLEEP
Node 17 -> Next Hop: 12
Node 17, Energy: 0.95, State: SLEEP
Node 18 -> Next Hop: 8
Node 18, Energy: 0.95, State: SLEEP
Node 19 -> Next Hop: 13
Node 19, Energy: 0.95, State: SLEEP

Visualization Output:

Red color : Base Station Node.

Blue color : Ordinary Node.

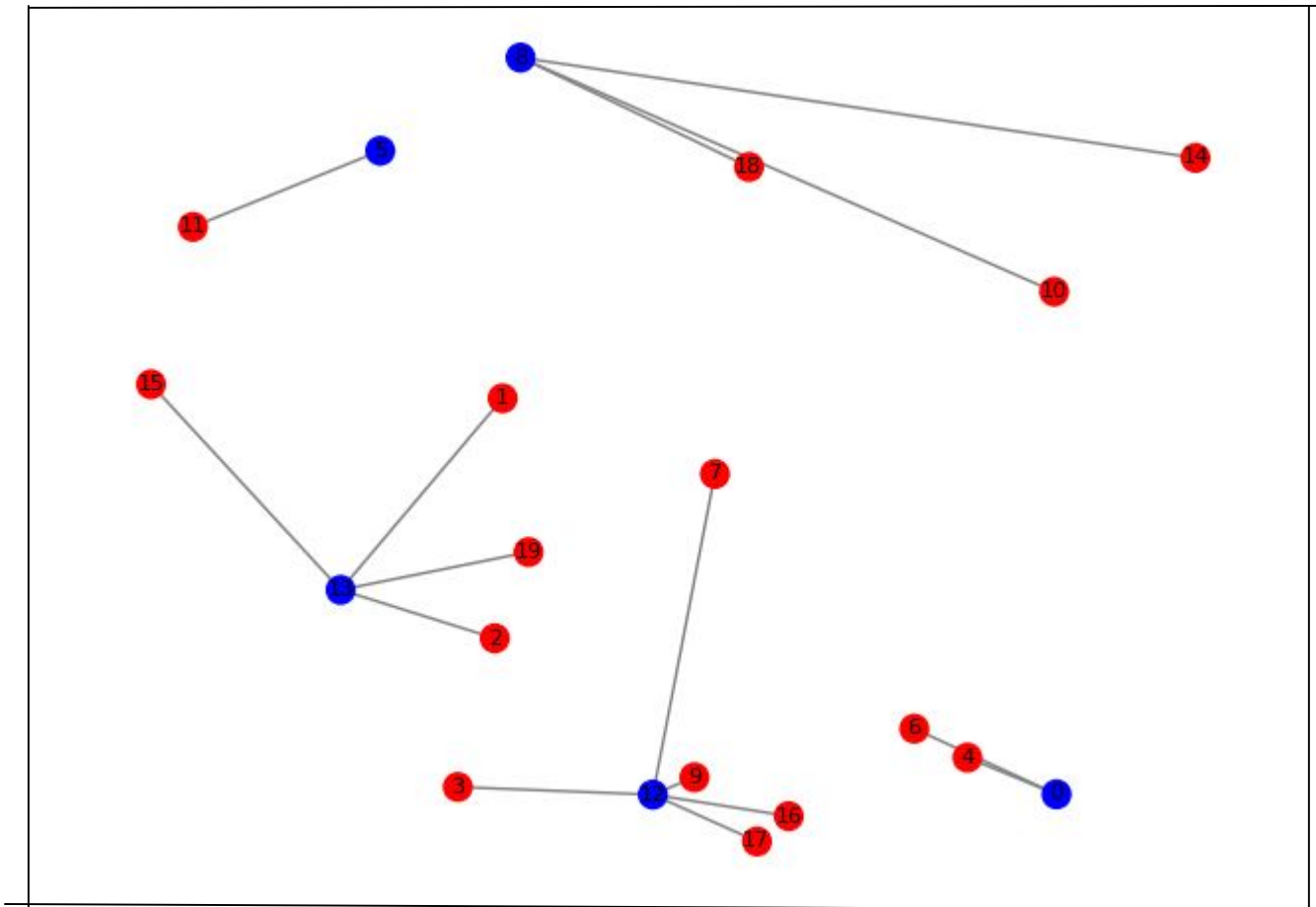


Figure : 14

----- END -----